Trabalho Prático

Gabriel Teixeira Carvalho

1. Introdução

Esta documentação lida com o problema de implementar um alocador de bicicletas. O objetivo principal desta tarefa é criar um programa que, dado um mapa e uma série de preferências das pessoas, consiga alocar bicicletas a pessoas de acordo com as preferências e as distâncias entre elas. Para resolver o problema citado, foram utilizados os algoritmos Breadth-First Search e Gale-Shapley.

A seção 2 desta documentação trata sobre a modelagem computacional do problema, enquanto na seção 3 são apresentadas as estruturas de dados e algoritmos utilizados e o pseudocódigo do sistema. Por fim, na seção 4 são apresentadas análises de complexidade assintótica de tempo, seguida por um manual de compilação e execução do programa.

2. Modelagem computacional do problema

A modelagem computacional desse problema parte da conversão de um mapa de N linhas e M colunas para um grafo com vértices numerados de 0 a N*M – 1 e arestas entre os vértices que não são obstáculos que estão acima, à direita, abaixo e à esquerda.

Além disso, esse grafo é representado como uma lista de adjacências para que seja o algoritmo de busca em largura, BFS, visando calcular a distância entre os vértices de interesse para todos os outros. Com esse resultado e com os pesos dados às bicicletas pelas pessoas, conseguimos criar listas de preferências de pessoas e de bicicletas que são utilizadas pelo algoritmo Gale-Shapley para gerar casamentos estáveis entre pessoas e bicicletas, de forma que estes casamentos sejam ótimos para as pessoas.

Essa modelagem pode ser entendida no exemplo abaixo:

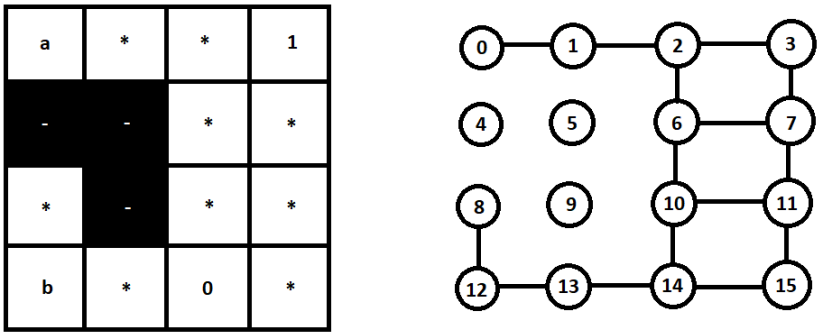

Figura 1: Mapa de caminhos, bicicletas, pessoas e obstáculos convertido para grafo

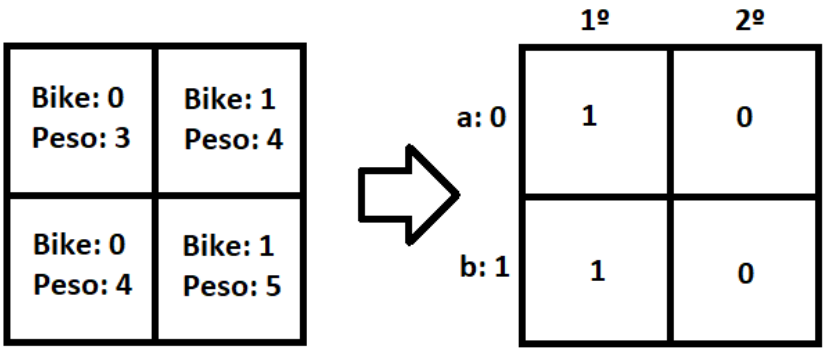

Figura 2: Conversão de pesos atribuídos pelas pessoas para preferências

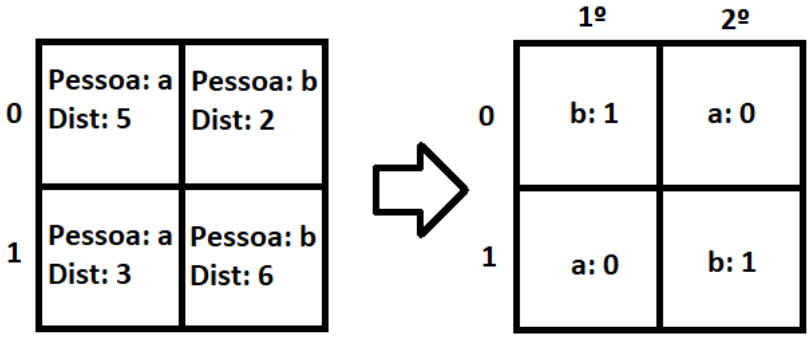

Figura 3: Conversão de distância das bicicletas para preferências

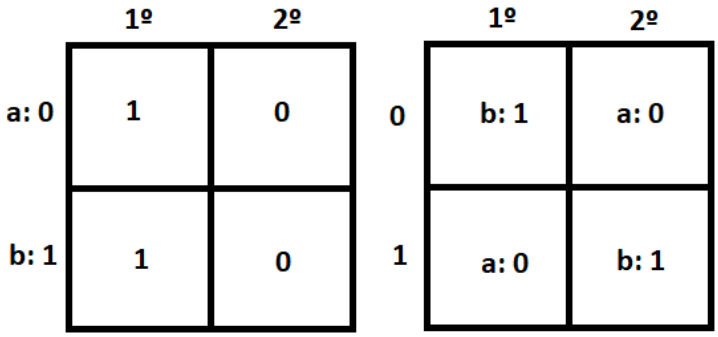

Figura 4: Problema de casamento estável

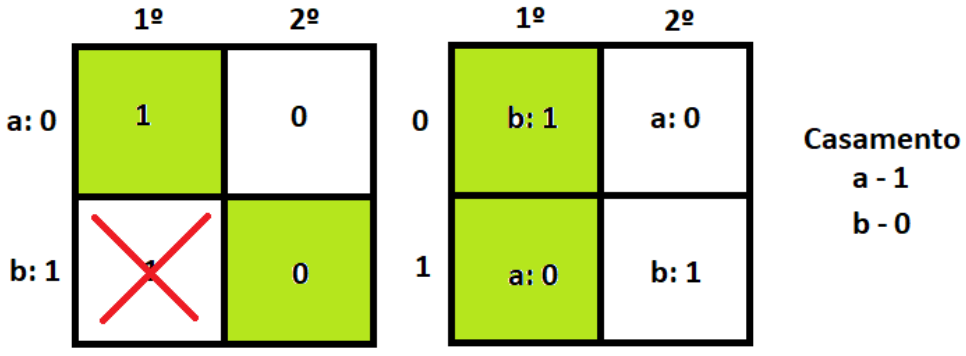

Figura 5: Solução do problema de casamento estável

3. Estruturas de dados e algoritmos utilizados

Neste trabalho prático utilizei das seguintes estruturas de dados e algoritmos:

- **Estruturas de dados:**

    - Structs: Utilizadas para armazenar em uma entrada de um vetor os ids e os pesos/distâncias para as pessoas e as bicicletas.
    - Vector (STL): Utilizado bastante ao longo do código para representar o mapa, matrizes de preferência, conteúdo dos vértices, etc. Muito útil pelo seu acesso rápido aos elementos e fácil utilização.
    - Pair (STL): Utilizado para representar os casamentos entre pessoas e e bicicletas.
    - Queue (STL): Utilizada para guardar e controlar as próximas pessoas a proporem no Gale-Shapley e o próximo vértice a ser analisado na BFS, por conta de seu comportamento First In First Out (FIFO).

- **Algoritmos:**
    - Breadth-First Search (BFS): Utilizado para calcular a distância de um vértice para todos os outros do mapa.
    - Gale-Shapley: Utilizado para fazer um casamento estável entre as pessoas e as bicicletas, levando em consideração as preferências das pessoas e as distâncias das bicicletas para as pessoas.
    - Algoritmo de ordenação (<*algorithm*>): Utilizado para ordenar as listas de pesos das pessoas em ordem decrescente de peso e crescente de ID. Também utilizado para ordenar as listas de distâncias das bicicletas em ordem crescente de distância e crescente de ID.

3.1. Pseudocódigo

```
Programa () {
  -    Recebe o número de pares
  -    Recebe o tamanho do mapa
  -    Recebe e salva o conteúdo dos vértices
  -    Cria lista de adjacências do mapa
  -    Recebe os pesos das bicicletas para as pessoas
  -    Converte pesos para preferências das pessoas
  -    Ordena as listas de pesos
  -    Converte e salva pesos ordenados como
  preferências das pessoas, convertendo id char para
  int pela tabela ASCII
  -    Calcula distâncias entre as bicicletas e as
  pessoas aplicando BFS
  -    Converte e salva distâncias para preferencias das
  bicicletas, convertendo id char para int pela tabela
  ASCII
  -    Ordena as listas de distâncias
  -    Salva distancias ordenadas como preferências de
  bicicletas
```

- Aplica o Gale-Shapley
- Imprime o casamento, convertendo int para char pela tabela ASCII

}

BFS(verticeInicial, listaDeAdjacencias) {

- Inicializa vetor de vértices descobertos como falso
- Inicializa vetor com distancias dos vértices do grafo ao verticeInicial. Inicia todos com distancia "Infinita"
- verticeInicial é descoberto e tem distancia 0
- Inicializa fila com os vertices a visitar
- Enquanto (fila não está vazia) {
    - Pega um vertice V da fila
    - Para cada vertice adjacente a V, se não tiver sido descoberto, adiciona ele na fila, define como descoberto e define sua distância como a distância de V ao verticeInicial + 1

  }
- Retorna o vetor de distancias

}

Gale-Shapley(preferenciasPessoas, preferenciasBicicletas, numeroDePares) {

- Inicializa vetor de casamento retornado pela função, com os pares alocados
- Inicializa vetor com o índice da próxima bicicleta à qual cada pessoa irá propor
- Inicializa vetor com os casamentos
- Inicializa fila das pessoas que ainda irão propor
- Inicializa matriz com a preferência da bicicleta i para cada pessoa j no elemento [i][j]
- Enquanto (houver pessoas sem par) {
    - Propõe para a próxima bicicleta em sua lista de preferências
    - Se a bicicleta não tiver par ainda, aloca
    - Senão, compara se a proposta melhora a situação da bicicleta. Se sim, aloca

  }
- Passa os casamentos para o vetor de casamento
- Retorna o casamento

}

4. Análise de complexidade de tempo

Dado que o número de linhas do mapa seja M, o número de linhas seja P e o número de pares pessoa-bicicleta seja N, a complexidade de tempo dos algoritmos usados no código são:

BFS: Cada análise de um vértice V adjacente a outro vértice consiste somente em operações O(1), tais como colocar V na pilha, defini-lo como descoberto e salvar sua distância em um vetor ou pular a iteração caso ele já tenha sido descoberto. Como o grafo possui M*P vértices e cada vértice está ligado a no máximo 4 outros vértices, o número de arestas é E <= 4*M*P. Como o grau de um vértice indica quantos vértices serão analisados a partir dele, somando o grau de todos os vértices do grafo chega-se à complexidade das análises, O(2*4*M*P) = O(M*P), pelo lema do aperto de mão (soma dos graus de um grafo é 2*(número de arestas do grafo)). Além disso, são necessárias M*P operações de passar por cada vértice e atualizar o vetor de distâncias. Portanto, a BFS possui complexidade O(M*P).

Gale-Shapley: Cada proposta de uma pessoa para uma bicicleta leva tempo constante pois, se a bicicleta não tiver par, são feitas somente operações O(1) para atualizar os vetores de casamentos e retirada da pilha. Caso a bicicleta já tenha um par, uma operação de comparação O(1) entre o ranking da pessoa propondo e da atual é feita e então os vetores de casamentos e a pilha são atualizados. Como existem N pessoas e N bicicletas, então existem N * N = N^2 possíveis pares pessoa-bicicleta. Como cada pessoa pode propor no máximo uma vez para cada bicicleta, a complexidade assintótica desse algoritmo é $O(N^2)$.

Sort: Esse algoritmo do header <algorithm> da linguagem C++ possui a complexidade ótima dos algoritmos de ordenação, Θ(N*log(N)). Como a lista de pesos e distâncias a serem ordenadas possuem N elementos cada, a complexidade desse algoritmo é O(N*log(N)).

Programa: Receber as entradas e converter o mapa para um grafo possui complexidade O(M*P). Ordenar as N listas de pesos de tamanho N das pessoas possui complexidade O(N*N*log(N)) = $O(N^2*\log(N))$. Calcular as distâncias com a BFS das N bicicletas possui complexidade O(N*M*P). Ordenar as N listas de distâncias de tamanho N das bicicletas possui complexidade O(N*N*log(N)) = $O(N^2*\log(N))$. Achar um casamento estável com o Gale-Shapley possui complexidade $O(N^2)$. Imprimir os resultados possui complexidade O(N). Portanto, o programa possui complexidade $O(M*P + 2*N^2*\log(N) + N*M*P + N^2 + N) = O(N^2*\log(N) + N*M*P)$.

Instruções para compilação e Execução

1 – Extraia o arquivo `.zip` na pasta desejada.

2 – Execute o comando `'g++ tp01.cpp -o tp01'` no terminal.

3 – Execute o programa `tp01` passando um arquivo de texto com a entrada para o programa pela linha de comando.